1

Productivity Engineering in the UNIX† Environment

SOAR: Smalltalk Without Bytecodes

Technical Report

S. L. Graham

(415) 642-2059

"The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government."

Calif Univ Berk.

Contract No. N00039-82-C-0235

November 15, 1981 - December 30, 1985

Arpa Order No. 4031

86 4041

†UNIX is a trademark of AT&T Bell Laboratories

86 10 3 069

SOAR: Smalltalk Without Bytecodes
(Preliminary Version)

A. Dain Samples
David Ungar
Paul Hilfinger

Abstract

We have implemented Smalltalk-80 on an instruction-level simulator for a RISC microcomputer called SOAR. Measurements suggest that even a conventional computer can provide high performance for Smalltalk-80 by :

- 1) abandoning the 'Smalltalk Virtual Machine' in favor of compiling Smalltalk directly to SOAR machine code,
- 2) linearizing the activation records on the machine stack;
- 3) eliminating the object table, and
- 4) replacing reference counting with a new technique called Generation Scavenging.

In order to adopt these techniques, we had to find new ways to support

- hashing objects,
- accessing well-known objects,
- invoking blocks,
- referencing activation records,
- managing activation record stacks, and
- converting the virtual machine images.

This work sponsored in part by Defense Advance Research Projects Agency (DoD) Arpa Order No. 4031 Monitored by Naval Electronic Systems Command under Contract No. N00039-82-C-0235.

It is so difficult to find the *beginning*. Or, better, it is difficult to begin at the beginning. And not try to go further back. - Ludwig Wittgenstein, *On Certainty*, §471.

1. Introduction

This paper focuses on software techniques to support Smalltalk[1] on conventional architectures. It reports on our experiences implementing Smalltalk On A Reduced Instruction Set Computer (SOAR). Although SOAR has some hardware support for running Smalltalk, our experience has led us to the conclusion that efficient execution of Smalltalk requires less hardware support than we initially supposed. This survey of our implementation should provide a roadmap for those wanting to implement Smalltalk on conventional architectures.

In the discussion that follows, we assume the reader is somewhat familiar with both ST-80 as defined in [Goldberg83], and with Reduced Instruction Set Computers as discussed in [Patterson85].

The designers of the Smalltalk programing language adopted the purist position that everything in the system would be an object. This was not limited to the usual basic data types, but extended even to the state of the machine: activation records, instructions, program counters all conformed to a specified format. For example, since the design did not countenance pointers into the middle of subroutines, the return address for every subroutine call and even the program counter had to be an integer offset, not an absolute address [Goldberg83]. Even the most frequently accessed of all data - instructions - were constrained by this design. The language was defined in [Goldberg83] in terms of an interpreter for a virtual machine with a set of instructions they called "bytecodes". This made Smalltalk portable since it is relatively easy to understand the virtual machine and to write an interpreter for it. Smalltalk was developed on research machines (the latest of which is the Dorado) that had writable control stores and could do the interpretation in firmware [Deutsch83] [Lampson81] [Thacker79]. So with the Dorado's 70ns micro-cycle time, they were able both to define a portable virtual machine for Smalltalk, and still achieve very acceptable performance interpreting the bytecodes.[2]

There are several reasons why Smalltalk programs have proven especially difficult to execute quickly.

- The language has been defined in terms of a bytecode interpreter. Interpreters are slow.
- The object oriented nature of the language implies a huge number of time consuming procedure calls, or "messages".
- The definition of Smalltalk execution requires the rapid creation and automatic reclamation of many objects. This puts a heavy demand on the memory management mechanism.

[1] Smalltalk-80 is a trademark of the Xerox Corporation. Whenever we are referring to Smalltalk as defined by Xerox in [Goldberg83], we will use 'Smalltalk-80', or its abbreviation ST-80. Whenever we refer to the *language* apart from any implementation, we will refer to it as simply 'Smalltalk'.

[2] In some circles, this is not called interpretation, but execution of native machine code.

Most of the early implementations of Smalltalk on 'traditional' von Neumann-type architectures have been evaluated as slow to abysmal [Krasner83]. Efforts to speed up interpreter execution have included predeclaration of object types [Suzuki84] and 'caching' Smalltalk procedures in native machine code while preserving the bytecode orientation of the definition [Deutsch84]. The SOAR project differs from the [Suzuki84] effort in that we do not require pre-declaration of types for efficient execution. We differ from [Deutsch84] in that we do not try to maintain the illusion of a virtual machine executing bytecodes. Abandoning this model of execution has forced our implementation to differ in several important ways:

- Our compiled methods are different: ST-80's contain bytecodes, ours contain integers (which are also SOAR machine instructions).
- Our method contexts (ST-80's activation records) are different: ours are fixed size, reside on the machine stack, and are moved into heap space only when necessary.
- Our block contexts are different: ST-80 blurs the distinction between a block and its activation record: we keep them separate.
- The *become* operation is being phased out and will eventually ceased to be supported.

These are discussed in more detail in the following pages. But these differences allow us to achieve acceptable performance executing Smalltalk.

The SOAR project started with the basic RISC assumption that memory would be plentiful and should be traded in exchange for speed. There were two questions: how can Smalltalk be executed quickly on more traditional architectures? and, what changes/enhancements to the traditional architectural model can produce a fast execution vehicle? The SOAR project concentrated on both questions with approximately equal emphasis, and our solution took the form of a judicious split of functionality between innovative hardware and software support. This paper will primarily discuss the project's answers to the first question. Answers to the second question can be found in detail in [Pendleton86] [Ungar84b] [Ungar86] and [Samples85]. Currently, we are still executing on a simulator running on Sun workstations. We have completely booted the system, it paints the windows, and we have run the macrobenchmarks. We are waiting for completely functional chips to plug into the board to run on Sun workstations. The migration path from the Xerox image to SOAR is outlined in Fig. 1. It is worth noting that we needed to use an existing Smalltalk system to modify the image. BS (Berkeley Smalltalk), an implementation running on Sun workstations that interprets bytecodes, provided that system.

The SOAR system is projected to run ten times faster than BS, about five times faster than the system described in [Deutsch84]: or about the speed of the implementation on the Dorado.

We will describe our software solutions in a bottom-up (inside-out) order. Thus, we start with general object management, including addressing and reclamation, and proceed to compiled subroutines, activation records (contexts), and processes. Table 2 summarizes the contents our presentation by listing problems we have considered and a brief outline of our solutions. The bytecode→SOAR compiler is outside the scope of this paper, see [Bush85] for more details.

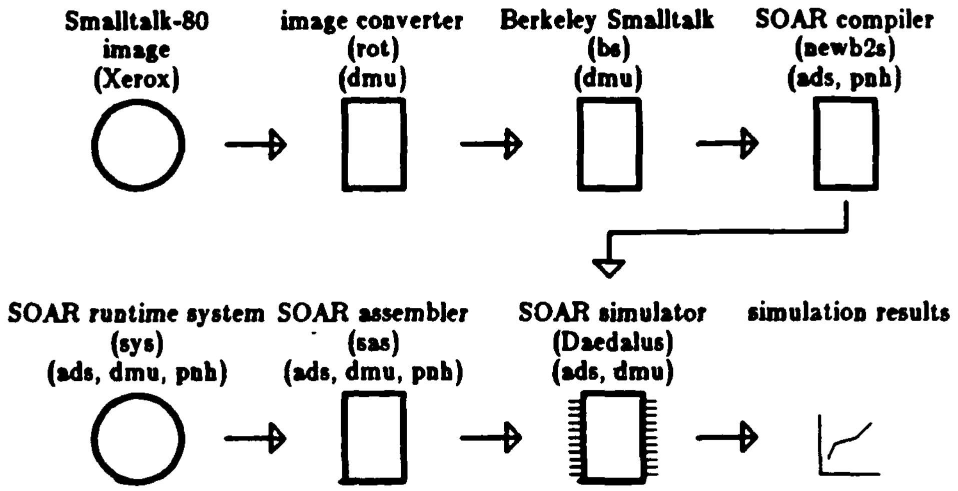
Smalltalk-80 image (Xerox)
image converter (rot) (dmu)
Berkeley Smalltalk (bs) (dmu)
SOAR compiler (newb2s) (ads, pnh)
SOAR runtime system (sys) (ads, dmu, pnh)
SOAR assembler (sas) (ads, dmu, pnh)
SOAR simulator (Daedalus) (ads, dmu)
simulation results

Figure 1: Steps involved in a SOAR simulation. First, *rot* removes the object table from the Xerox ST-80 image. We then use BS to make any modifications necessary in the image (e.g. to eliminate some *becomes*). BS (Berkeley Smalltalk) is a 68000 version of ST-80 which maintains the bytecode virtual machine as its lowest level: it was not a system noted for speed. Newb2s produces a Smalltalk image for SOAR by converting the BS objects to SOAR format and running Hilfinger's *Slapdash* compiler, which translates the bytecoded programs to SOAR instructions. We have also coded the Smalltalk primitive operations and storage management software in SOAR assembly language. After this is assembled, it is fed to our SOAR simulator along with the Smalltalk image. The initials below each system indicate its author: ads is Dain Samples, pnh is Paul Hilfinger, and dmu is David Ungar.

Table 2: Summary of problems and solutions

- **object addressing**: direct pointers, removing Object Table
- **become *sans* Object Table**: Rewrite system classes to use explicit indirection. Rejuvenation for new-old, copy for == length objs.
- **hashing *sans* Object Table**: extend header
- **accessing well-known objects**: Registry of needed objects; system routines for searching Smalltalk dictionaries.
- **storage reclamation**: Generation Scavenging: stop and copy young survivors
- **memory fragmentation**: Scavenging compacts young objects. Offline reor~~ganization of old objects~~. Paging of old objects.
- **efficient creation/deletion for activation records**: use stack of activation records for normal cases
- **pointers to activation records**: Detect non-lifo activation records by checking stores and return values. Maintain table of pointers to activation records in stack. On return from non-lifo context, copy it to heap, update pointers by searching table, and remove entry from table. Put object headers in gaps between activation records so activation record on stack looks like object.
- **block context objects**: Separate block prototype from its activation record. Block prototype is a real object, activation record is just activation record. Create proto upon blockCopy (but need not), create activation record upon entry.
- **non-local returns**: Rescue non-lifo contexts in a subroutine.
- **allocating activation record stack**: Resume primitive checks for blockContext.
- **freeing activation record stack**: Suspend primitive checks for call from terminate – rescues non-lifos and reclaims stack.
- **fast method lookup**: In-line cache
- **in-line cache invalidation**: Zapping old methods.
- **converting context objects in image**: Throw them away – use genesis method.
- **updating pcs into methods if method is scavenged**: methods must be old
- **debugger**: designed and coded, not yet put into use
- **compiler**: see [Bush]

2. Direct Addressing – Eliminating the Object Table

ST-80 addresses objects by an integer index into an object table, instead of through a direct word or byte address. Few real machines have the ability to perform such segment-oriented addressing on 100,000 segments averaging 14 words in length. Those that can - such as the iAPX-432 - pay a large price in speed or cost or both. In Smalltalk, the cost of indirection is justified by the need for cheap memory compaction: when objects are moved, only the object table need be updated. However, we did not want to penalize every object access just to ease compaction, so we eliminated the object table by designing a reclamation algorithm that also compacts. BS and SOAR are the only Smalltalk systems without object tables (Figs 3 and 4).

The indirection through such a table is an indirect cost of other storage management strategies that is sometimes overlooked. It can be a bottleneck: a typical ST-80 system accesses the object table 1.2 times per bytecode [Ungar83]. Assuming SOAR performs as fast as the Dorado (300K bytecode/sec), SOAR would access the object

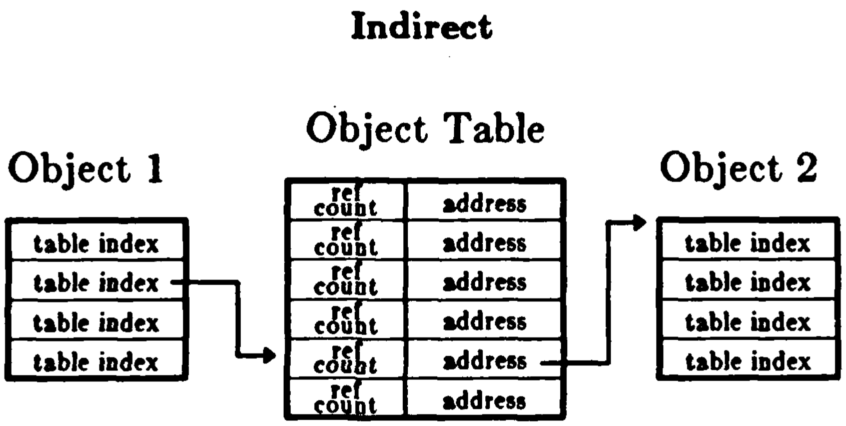

Figure 3: Indirect addressing. In traditional ST-80 systems, each pointer is really a table index. The table entry contains the target's reference count and memory address. This indirection required previous ST-80 systems to dedicate base registers to frequently accessed objects. The overhead to update these registers slowed each procedure call and return.

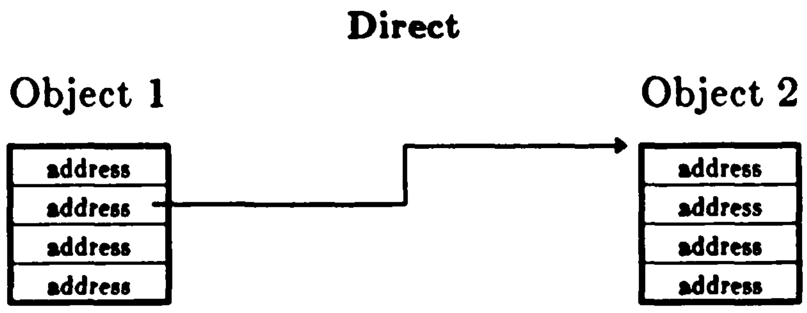

Figure 4: Direct addressing. A SOAR pointer contain the virtual address of the target object. This is the fastest way to follow pointers.

table 360,000 times per second. The absolute minimum table access would be a single load instruction, which takes two cycles. Assuming 400 ns per cycle, such an indirection would take 800 ns and, at 360,000 table accesses per second, that would be 0.29 seconds of indirection time for each second of processing time. Discussions with Deutsch suggest that further optimization possibly could halve this overhead. In other words, an object table would slow SOAR by 15% to 29%.

We have also estimated the impact of indirection on code size [Ungar86]. An Object Table would require an extra instruction to load or store a literal variable, and one indirection in the method prologue (for the receiver). We assume that many indirections will be optimized away, as in Deutsch and Schiffman's system, and that the Object Table can reference as many objects as can a direct-pointer system. Table 5 presents our analysis under these assumptions (i.e. 32-bit words). The extra code for an object table would add 2% to the size of the system.

Becomes. Although we eliminated the object table to improve performance, there is one ST-80 primitive operation that runs much slower without it. The *become:* primitive exchanges the identities of two objects, so that all pointers to the first object are redirected to the second, and vice versa.

A ST-80 system with an object table can perform a *become* quickly by exchanging object table entries (Figure 6). A system without an object table (such as SOAR) must search objects and exchange pointers. Although we have devised strategies to limit the search, a worst case *become* still involves a search throughout virtual memory. Such a long pause is unacceptable. We avoided this problem by rewriting the software for ST-80 data structures to avoid *becomes*. To establish the feasibility of this approach, we added new Collection classes that mimic old ones without resorting to *becomes* (Figure 7), then modified the macro-benchmarks to take advantage of our *become*-less classes [Wallace83]. Table 8 presents an analysis of this change on system performance. The printDefinition benchmark shows that this change has a negligible effect on a benchmark that does not do any *becomes*. But, our efforts to eliminate *becomes* from programs that did use them were handsomely repaid with an 18% to 28% performance improvement.

Although we have eliminated *becomes* invoked by the system classes, the SOAR programmer must either shy away from this primitive, or be prepared to pay a stiff performance penalty [Ungar86]. However, we believe that the *become* primitive is so intrinsically expensive - requiring either a scan of virtual memory or a level of indirection that slows down many frequent operations - that alternatives should be sought. Eventually, all instances of *becomes* will be removed from the system, and the primitive no longer supported. The SWAMP project has reached the same

Table 5: Static cost of object indirection.	
method prologues	4654
literal variable loads	3532
literal variable stores	254
total image size	1,500 kB
relative cost of additional code	2.25%

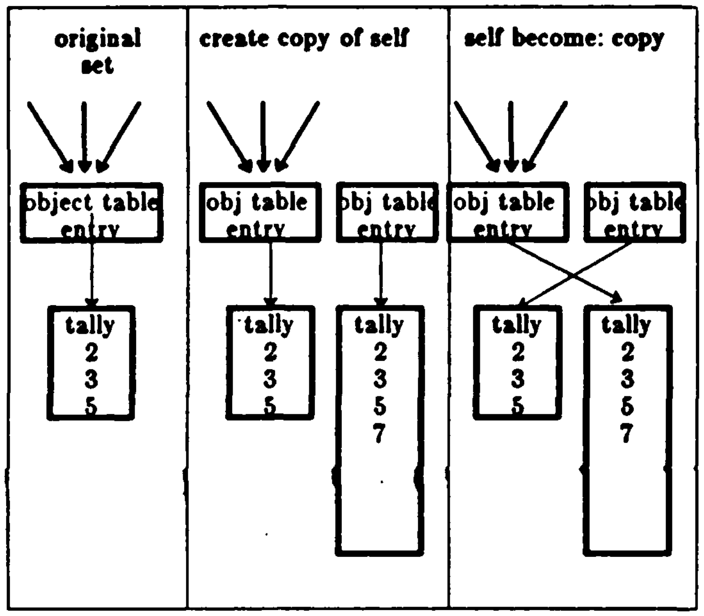

Figure 6: Growing with become. The sequence above illustrates how a set employs become to grow in a ST-80 system. Initially, the set is {2, 3, 5} and we attempt to add 7 to it. The set creates a larger copy of itself and uses become: to replace the original set with the larger version.

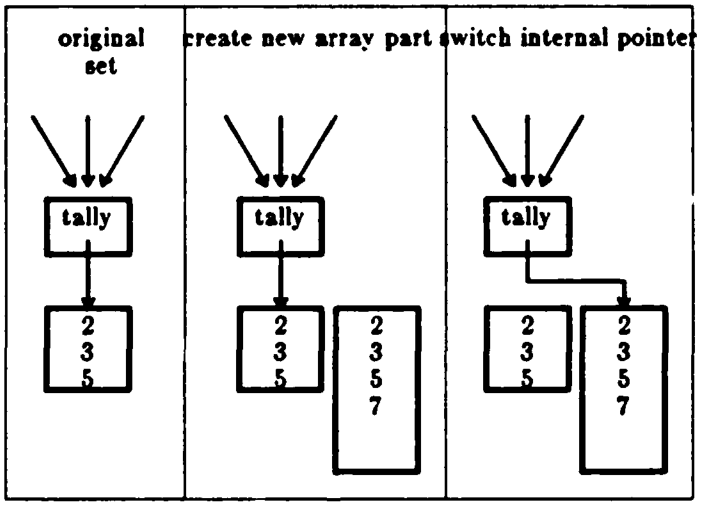

Figure 7: Growing without become. The sequence above illustrates how our modified sets grow without resorting to become:. The contents are stored in a separate array. To grow, the set allocates a larger array, initializes it, and redirects an internal point to the new array. We have replaced costly implicit indirection with explicit indirection that incurs cost only when needed. This is in keeping with the RISC philosophy.

Table 8: Performance impact of eliminating becomes.

benchmark	# becomes	duration w/ becomes (cycles)	duration w/o becomes (cycles)	cycles saved
printDefinition	0	75,475	75,317	0%
compiler	7	1,383,201	1,127,658	18%
decompiler	38	4,045,641	3,006,974	26%
printHierarchy	3	165,997	119,574	28%

conclusion [Lewis86].

Although we have eliminated the object table, there remain a few objects, such as nil, true, false, Point, String etc that the runtime support routines need to be able to find. This is almost the same set of objects with permanently assigned object table indices in [Goldberg83]. Our solution to this problem is to create a new "system object" that contains a table of pointers to well-known objects, as well as other necessary system data. Then the system object and the objects for nil, true, and false are locked down by assigning them to fixed locations in memory. [3]

Hashing. Perhaps the subtlest problem with eliminating the object table arises from the hashing semantics expected of Smalltalk objects [Kaehler]. Smalltalk requires that each object be capable of returning a hash value which is used to maintain ubiquitous hash tables (Dictionaries). The operation producing this hash value must therefore be efficient. Since we assume that time is critical and space is plentiful, we chose to add a field to the header of each object, containing an integer assigned by the instantiation primitive (*new* and *new:*). The hash primitive simply returns this field.

3. Generation Scavenging

Early in the SOAR project, we realized that automatic storage allocation and reclamation could easily become a bottleneck. We knew that the overhead for allocation and freeing in ST-80 systems ranged from 10% to 20% [Deutsch84] [Ungar83], that some reclamation algorithms introduced annoying pauses, that some required the programmer to explicitly free circular structures of objects, and that most of the algorithms had been implemented in microcode. Since we wanted to attain good performance without microcode, we designed, implemented, and measured *Generation Scavenging*, a new garbage collector that

- limits pause times to a fraction of a second,
- requires no hardware support,
- meshes well with virtual memory,
- reclaims circular structures, and
- uses only 3% of the CPU time in SOAR. This is less than a third of the time of deferred reference counting, the next best algorithm.[4]

[3] Our system needs to know the location of eighteen objects in heap space.

[4] Experience with SOAR has made us realize that some of the other algorithms that are usually microcoded need not be.

The details of Generation Scavenging have been reported elsewhere [Ungar84a] and [Ungar86]. Briefly and simply, memory is divided into two regions: one to contain old objects, the other containing new objects. All objects are allocated out of new space, and when this space is depleted, the live new objects are traversed and copied (see Fig. 9). Since most objects die young, there are relatively few objects that survive: on the average, only 3% to 5% of new objects survive and have to be moved [Ungar84a]. The algorithm actually used is more sophisticated than that just described, but nevertheless executes with less overhead that mark-and-sweep or reference counting algorithms (3% vs 9-20%).

The important point about Generation Scavenging in this context is that it places very little burden on program execution. For example, store instructions that put pointers to new objects into old objects must update a table. This cross-generation check can be accomplished with few instructions and no extra data references either in software by checking the two values against the new/old dividing line, or in hardware with a few tag bits in the address field, as in SOAR. It turns out that doing the check in software is so simple and infrequent that it was a mistake to have put the check into SOAR's store instruction. It can be done in software with a 1% performance penalty [Ungar86]. Furthermore, storing a young pointer into an old object is so rare (only 4% of all stores) that recording it adds only 0.05% overhead [Ungar86]. Implementing only a small number of generations which allows a cheap software check contrasts with, for example, ZetaLisp's strategy in which extra hardware in the page map is needed to keep track of many small generations and to

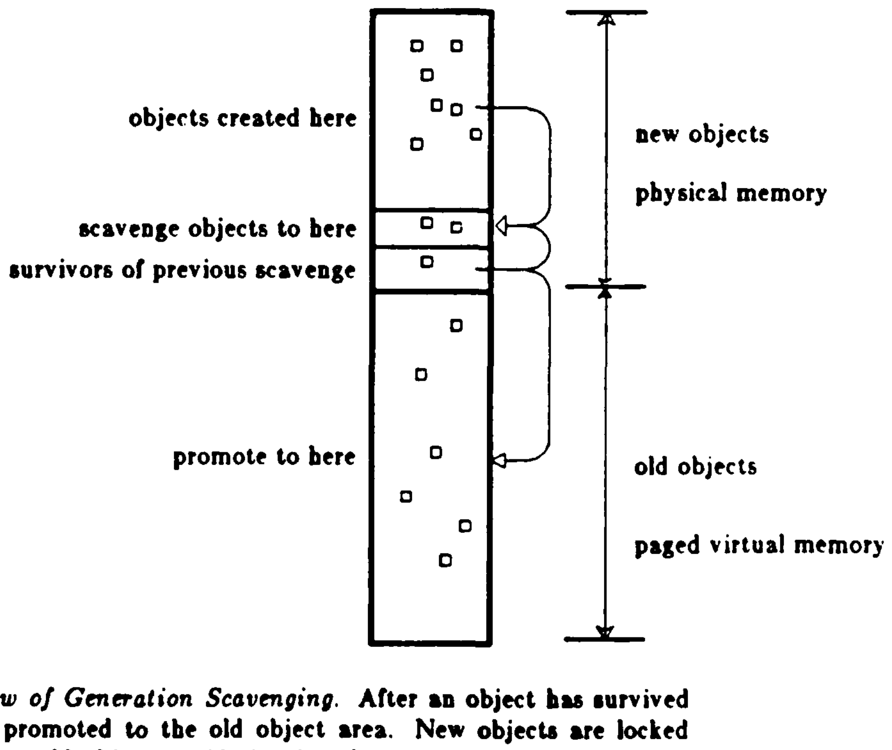

Figure 9: Bird's eye view of Generation Scavenging. After an object has survived enough scavenges, it is promoted to the old object area. New objects are locked down in physical memory; old objects reside in virtual memory and may be paged out.

check stores [Moon85].

4. Activation record management

Everything in Smalltalk is an object, and that includes activation records. If Smalltalk were implemented straightforwardly from the description in [Goldberg83] (and most of the systems described in [Krasner83] were initially so implemented) then each procedure call (message send) would require the allocation and initialization of an activation record from the heap. Each return would leave a dead activation record for garbage collection to reclaim. Given the high percentage of procedure calls in Smalltalk programs (83% of all objects allocated in ST-80 are activation records, and 28% of all bytecodes executed result in calls of one flavor or another [Falcone83]) activation record allocation and freeing can be a very large proportion of the running time. So some sort of blending of Smalltalk's model of activation records (context objects) and an efficient stack implementation is needed for smooth execution [Waring]. This union seems particularly inviting in view of the fact that, in spite of the "full object" citizenship Smalltalk accords to activation records, 85% of them behave like traditional activation records during their lifetime: they are created by a call, never used as data objects, and are released as soon as the executing procedure (method) exits [Deutsch84].

This problem has been attacked in other Smalltalk systems on conventional machines. [Suzuki84] keeps a small memory area in which to keep the machine stack. When the stack becomes full, activation records are swapped out to heap space in first-in-first-out order (much like managing register windows on a RISC chip). If any context in the stack needs to be retained then all contexts are swapped to heap space. [Deutsch84] creates a context object either in heap space, or on the machine stack, depending on how and when the object is created. If a pointer is generated to a context on the machine stack, then it is marked specially to be popped into heap space instead of oblivion. The Tektronix system [.can't find our ref.] caches the current context in a convenient format, and eliminates the allocation and initialization for leaf activation records.

Our approach varies in two ways: we have a more selective algorithm for detecting those activation records that need to be moved to the heap, and we don't try very hard to mask the differences between ST-80 contexts and our activation records. Objects in heap space can point to contexts still on the stack. Figure 10 illustrates how registers are stored in the stack with 'gaps' between the activation records: the gaps contain the context object headers. The main difference between our implementation and ST-80 is that we support only one size of activation record: the SOAR stack frame is sixteen words. Once this change is promulgated throughout the system, it doesn't make any difference to objects manipulating activation records whether the AR's are on the stack or in heap space.

But now we have the problem of knowing when an activation record on the top of the machine stack can be discarded or must be moved into heap space. For example, a subroutine can obtain a pointer to its own activation record and place it in a global variable. After the subroutine returns, another routine can inspect the activation record via the global variable. In this case, it is necessary to have moved the activation from the machine stack into heap space. (See Fig. 11.) Extraordinary measures are required to preserve the correct objects. Our strategy is like that for

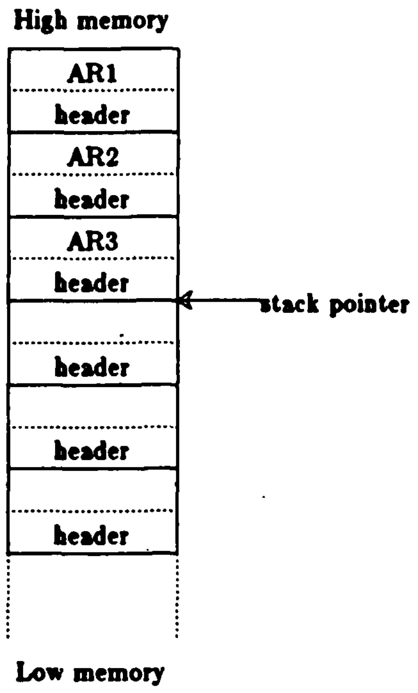

Figure 10: SOAR Activation records. SOAR activation records showing the gaps in memory in which the object headers for the context objects are placed. The diagram shows three active contexts in the stack.

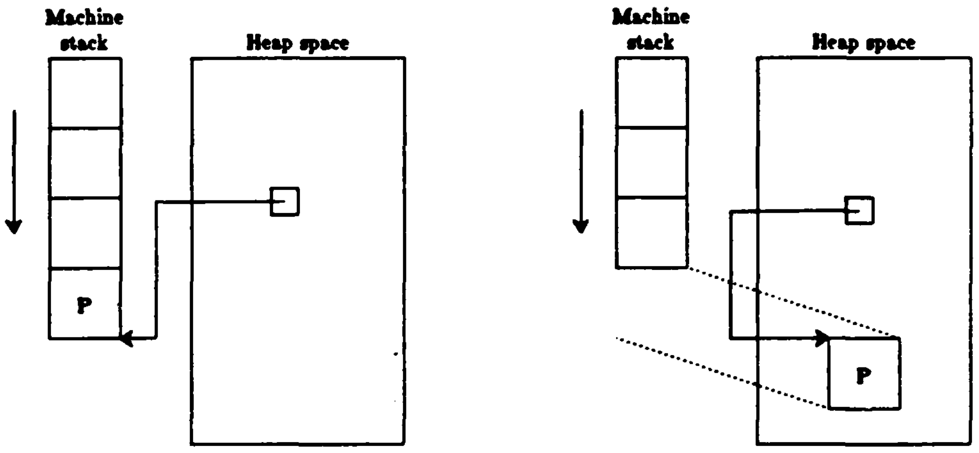
Machine stack
Heap space
P
Machine stack
Heap space
P
P stores a pointer to its activation record
When P returns, its activation must be moved to Heap space because of this pointer.

Figure 11 Saving activation records

generation scavenging: we monitor stores and returns. When a pointer to an activation record is either stored into an object or returned up the call stack the referenced activation record is marked as *non-lifo:* i.e., the activation record is likely to outlive its existence on the LIFO machine stack. When a *non-lifo* activation record is about to be destroyed (i.e. when a return instruction would pop it off the stack) it must be moved from the stack to the heap. Thus, the steps are:

(1) Check for stores of pointers to activation records and for activation records as return values. Mark any such activation records as non-lifo.

(2) On returns, check to see if the current activation record has been marked. If so, the return must move the activation record and adjust pointers to it.

Blocks. Activation record management becomes much more complicated when Smalltalk blocks (*aka* funargs, closures) are implemented. ST-80 blocks implement control structures by allowing one routine to control execution in another's context. Frequently, a block is created, passed down the call chain to a subroutine that repeatedly invokes the block and then returns. Thus, we must impose a minimum of overhead on block implementation, while handling the non-lifo references. Thus, we do not mark an activation record as non-lifo if the only references to it are from blocks. Instead, the store and return checks treat the block as a surrogate for its home activation record. If a pointer to a block is stored, its home gets marked as non-lifo. In other words, although a block is an object that refers to a context, we do not mark the context as non-lifo until the block itself becomes non-lifo.

We differ from [Deutsch84] in that, apparently, they create activation records for Smalltalk blocks automatically in heap space, while we treat them as stack allocated activation records. Without going into excruciating detail, we would criticize the design of Smalltalk on this point: the Smalltalk notion of a block object folds the notion of an activation record for that block into the block object itself. That is, while Smalltalk separates the notion of a procedure and its associated activation record(s), the separation between blocks and their activation records is muddled. In our implementation, we separate these inappropriately confused notions, and keep activation records for blocks on the machine stack along with activation records for procedures. The blocks are objects in their own right. (Peter Deutsch put this bee in our bonnet.)

Any time a new activation record is allocated on the stack, all entries in that record must be initialized to nil to avoid dangling references (see Fig. 12). This can be an unacceptable overhead if one is using fixed length records that may not be completely utilized by all activations.

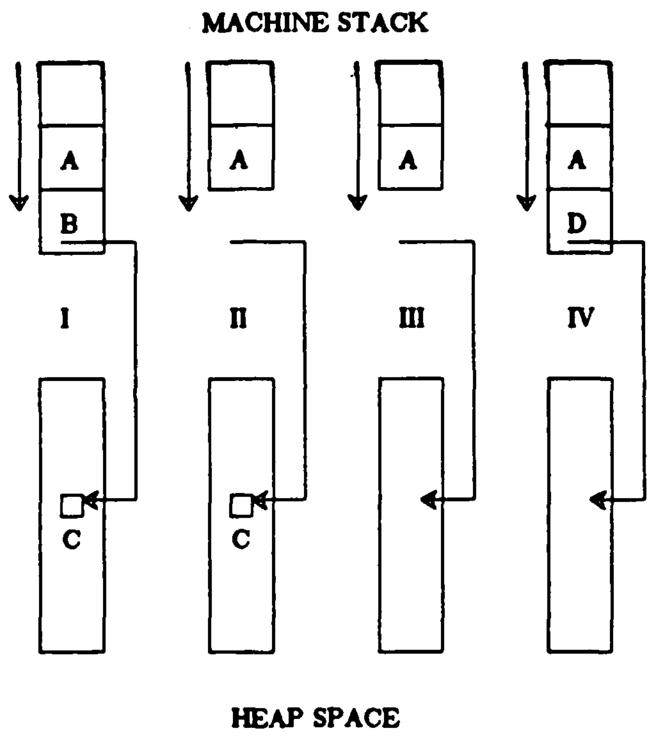

Figure 12: Machine stacks and scavenging. (I) Procedure A calls procedure B. Procedure B creates a pointer to object C in its activation record and (II) returns. At this point a garbage collection takes place and (III) reclaims the memory occupied by the now useless object C. (IV) Procedure A now calls procedure D, which does not reinitialize the register containing that old pointer to C. If another garbage collection were to occur at (IV), the activation record for D would contain a pointer to an object that no longer exists. If no garbage collection were done between (II) and (IV), then C would be retained by a garbage collection at (IV). The method described in the text using a *highwater mark* would have nilled the pointer to C during the garbage collection at (II).

We implemented a technique whereby we avoid this situation without having to completely initialize each and every register and word of an activation record on procedure entry. As procedures are invoked, a record is kept of the deepest point reached on the stack; on SOAR, keeping track of this *highwater* mark is easily done as part of the window handler. When generation scavenging occurs, the active portion of the stack is scanned and all portions of the stack between the current activation record and the highwater mark are nilled. Note that while objects may be needlessly scavenged, (again, see Fig. 12) they are guaranteed to exist. By initializing only those portions of the activation record that are used, and keeping track of the highwater mark between generation scavenges, we eliminate any possibility of dangling references, and obviate the need to initialize entire activation records.

One problem with our modification of the format and handling of activation records is that the ST-80 debugger that comes with the image from Xerox PARC can no longer be used. Debugging is only slightly more complicated for SOAR code than with bytecodes. It certainly is no worse than any other machine code debugger, and, because it will be embedded in a Smalltalk system, it will certainly be a 'symbolic' debugger: the user should never see SOAR machine code (unless, of course, she wants to). While decompiling SOAR code to Smalltalk may be difficult, keeping the source code memory resident removes the difficulty. Because Smalltalk procedures are small, and compilation is fast, an error location in a sequence of SOAR instructions can be quickly mapped onto the appropriate location in the Smalltalk routine by simply recompiling the routine where the offense occurred. (The Turbo-Pascal system makes very effective use of this technique [Borland86].) We have a debugger implemented for Smalltalk on SOAR, although it has not yet been incorporated into our system and itself debugged. There are still open questions in this area, and work on the debugger remains in progress.

5. Process Management

In ST-80, processes are simply objects that point to the current activation record, which in turn points to the preceding activation, etc. When the process object is no longer referenced, it is reclaimed, and all activation records for that process are also reclaimed if not referenced elsewhere. In SOAR, activation records are not moved into heap space except under certain conditions. Special measures must be taken to initialize processes and activation records, and to reclaim the memory they use when they are no longer active.

On SOAR, each process is allocated a fixed size chunk of memory for its (machine) stack (its activation record stack). When there are more activations in a process than can fit in this memory, a new chunk is allocated and the activation record stacks are linked. Managing activation record stacks works well on SOAR because of mechanisms that exist for RISC register windows [Katevenis83]. Whenever a call (or message send) would deplete the number of register windows available on the chip, a branch is taken to a window handler. The handler spills the oldest window on the chip into the activation record stack, and also checks for activation record stack overflow. If the activation record stack is also about to overflow, then the remaining register windows still on chip are spilled into the current activation record stack, a new activation record stack is allocated and initialized, and execution begins in the new activation record stack. When a return instruction is interrupted by a window underflow (i.e. the appropriate activation record is not on chip), a check is made for activation record stack underflow, and the inverse operation is performed.

Images. Changing instruction sets presents another problem related to processes. Smalltalk images are saved states of execution. There is no such thing as a bootstrap of Smalltalk because all images are frozen snapshots of executing systems, including any active processes. Sometime in the early 1980's one version of Smalltalk was booted and initialized, and all versions since have been snapshots of that original boot. We refer to this as being saved, but not born again.

Because we were throwing away bytecodes, linearizing activation records on a machine stack, and changing the garbage collection method (among other things), we determined that converting a running image was much more work than simply booting a re-compiled system. We then had to determine how to boot and initialize the system. When we asked Peter Deutsch about this possibility, he responded that it had been so long since anyone at Xerox had tried, he didn't know if the initialization code worked any longer. Fortunately for us, the code in the Smalltalk system for booting was correct. We wrote one procedure that would call and initialize all of the appropriate objects and begin spinning off the necessary background processes. The procedure consisted of less than a dozen lines of Smalltalk, and we called it "genesis".

Dead processes. There was another consequence of our decision to "stackify" the Smalltalk activation records: we could no longer depend on automatic storage management to reclaim dead processes. The problem occurs principally because the Xerox Smalltalk implementation simply suspends processes and depends on reference counting to reclaim the storage; hence, it does not provide a "terminate process" primitive. While generation scavenging reclaims the process objects themselves, the activation record stacks are *not* Smalltalk objects in heap space and must be treated differently. But the lack of a 'process terminate' means it is difficult to tell when activation record stacks can be reclaimed. We have solved this problem by having our version of the "suspend process" primitive check all process suspensions in the context of the call to determine if it really is a suspension or an effective process terminate: the process stack can then be reclaimed immediately.

6. Results

Anyone interested in implementing Smalltalk efficiently needs not only to understand the dynamics of Smalltalk as defined in [Goldberg83] and measured in [Krasner84], but also the potential consequences of modifying that definition. In Table 13 we present figures from our simulations of our system running the standard macrobenchmarks[5] used in [Krasner84] to judge the execution efficiency of Smalltalk implementations.

One conclusion is apparent: the system primitives are important. For every second spent in compiled code, three seconds are spent in the runtime system. This is best explained by the fact that, even for a compiled system, much of the action still occurs in those routines that define the execution of the primitives on which the Smalltalk bytecodes depend.

Table 13. [from Ungar86, App. A & B].

Benchmark	time in runtime library	time in cache check	total time in runtime system
classOrganizer	59%	14%	73%
compilerBenchmark	66%	10%	76%
decompiler	68%	10%	78%
printDefinition	62%	11%	73%
printHierarchy	76%	7%	83%
average	66%	10%	76%

[Ungar86] concludes that of all the features designed into the system, the software features turned out to be more important than the hardware features in their effect on the final performance figures. The speedups attributable to specific features of the hardware and software are summarized in Table 14. This table indicates how much longer a task would take if the indicated feature were removed from our system. For example, we use the in-line cache presented in [Deutsch84]. If we removed the in-line cache, our system would run 26% slower. If all the features indicated were removed, a task that currently takes 100 seconds would then take 263 seconds.

Table 14. Software vs. Hardware improvements [from Ungar86].

Software	158%
compilation (estimated)	100%
in-line cache	26%
direct + GS	32%
Hardware	105%
register windows	46%
tagged integers	33%
non-delayed jumps	11%
single cycle nilling of activation record	4%
software interrupt	7%
trap instructions	4%

The SOAR project took the well-worn phrase "hardware prices are falling" at face value and assumed that users would rather spend money on memory chips than on complicated mechanisms. But what exactly is the cost? Compiling Smalltalk into SOAR increases the size of the Smalltalk image about 0.5 Mb. Peter Deutsch estimated that compiling all of the Smalltalk image into 68000 code would increase its size by one megabyte [Deutsch84]. Given that the original Smalltalk image is over 1.5 Mb, it appears that compiling to SOAR is a reasonable tradeoff.

Table 15 contains information on the amount of code that had to be written to implement Smalltalk on SOAR. The C code running on the Sun includes the interface routines with the SOAR board, interface routines with the Sun's graphics display device[6], floating point, and file system interface routines. It does not include the code for Bill Bush's Smalltalk→SOAR compiler written in Smalltalk, nor for the code in "genesis".

[5] To be contrasted with the microbenchmarks which check the efficiency of the more primitive facilities of the system (e.g. plus, array reference, string concatenation, etc).

[6] For Smalltalk aficionados, note that BitBlt is on the Sun side, but CharacterScanner is on the SOAR side.

Table 15: Breakdown of code for SOAR runtime system.

files	lines	words	chars	function
19	3365	12491	90094	misc files
16	3847	11799	95817	prim files
2	1126	4845	34141	trap handler files
3	364	1604	11063	process files
40	8702	30739	231115	SOAR sub-total files
22	3134	9909	73518	Sun interface files (in C)
66	11836	40648	304633	grand totals

7. Conclusions

A bytecode virtual machine is very effective for defining precisely the semantics of a language, and bytecode interpretation is an exceptionally fast way of obtaining a ~~slow~~ ~~implementation~~ of that language. The SOAR project has confirmed that compiling directly to native code on a RISC architecture is a viable implementation route for a fast implementation. Our experience has also confirmed that it is possible to compile to the native code of more traditional von Neumann architectures and achieve reasonable performance [Deutsch84].

We have also confirmed that memory can be traded for performance. Smalltalk on 400ns SOAR will run as fast as the fastest known implementation, the Xerox Dorado.

8. Acknowledgements

Many of the ideas in this paper are refinements of, direct theft of, or reactions against ideas batted around in conversations with Peter Deutsch: his inspiring influence is gratefully acknowledged. We would also like to thank Bill Bush for writing the Smalltalk version of the Smalltalk→SOAR compiler and for many discussions on design issues; Joan Pendleton, Will Brown, and Frank Dunlap for help in (almost) mastering board schematics; and, of course, Dave Patterson for supporting all of us.

9. Bibliography

[Borland86] *Turbo-Pascal*, Borland International, Scotts Valley, Ca., 1986.

[Bush85] William R. Bush, "Smalltalk-80 to SOAR Code", Master's thesis, University of California at Berkeley, CS Dept., Nov. 1985.

[CS292R] *Smalltalk on a RISC, Architectural Investigations*, Proceedings of CS292R, April, 1983.

[Deutsch83] L. Peter Deutsch, "The Dorado Smalltalk-80 Implementation: Hardware Architecture's Impact on Software Architecture, " in *Smalltalk-80: Bits of History, Words of Advice*, G. Krasner (editor), September, 1983, pp 113-126.

[Deutsch84] L. Peter Deutsch, Allan M. Schiffman "Efficient Implementation of the Smalltalk-80 System", 11th POPL, Salt Lake City, Utah, 1984 pp 297-302.

[Falcone83]	Joseph R. Falcone, "The Analysis of the Smalltalk-80 System at Hewlett-Packard", in *Smalltalk-80: Bits of History, Words of Advice*, G. Krasner (editor), September, 1983.
[Kaehler]	Private communication.
[Krasner83]	G. Krasner, *Smalltalk-80: Bits of History, Words of Advice*, Addison-Wesley, Reading, MA, 1983.
[Katevenis83]	Manolis G.H. Katevenis, *Reduced Instruction Set Computer Architectures for VLSI*, PhD thesis, University of California, Berkeley, October 1983.
[Goldberg83]	A. Goldberg, D. Robson *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, MA, 1983.
[Lampson81]	Butler W. Lampson, "The Doradoh: A High-Performance Personal Computer", Xerox PARC Technical Report CSL-81-1, Jan. 1981.
[Lewis86]	David M. Lewis, David R. Galloway, Robert J. Francis, Brian W. Thomson "Swamp: A Fast Processor for Smalltalk-80," unpublished manuscript. [added note: this manuscript has subsequently been submitted to this conference]
[Moon85]	D. A. Moon, "Architecture of the Symbolics 3600," *Twelfth Annual International Symposium on Computer Architecture*, Boston, MA, June 1985, pp 76-83.
[Patterson85]	David Patterson, "Reduced Instruction Set Computers," *Communications of the ACM, 28(1) Jan. 1985, pp 8-21.*
[Pendleton86]	Joan Pendleton, "A Design Methodology for VLSI Processors", PhD thesis, Dept. of EECS, University of California, Berkeley, Sept. 1985.
[Samples85]	Dain Samples, Mike Klein, Pete Foley, "SOAR Architecture", Computer Science Division (EECS), University of California, Tech. rep. UCB/CSD 85/226, March 1985.
[Suzuki84]	Norihisa Suzuki, Minoru Terada "Creating Efficient Systems for Object-Oriented Languages", 11th POPL, Salt Like City, Utah, 1984, pp 290-296.
[Ungar83]	David Ungar, David Patterson, "Berkeley Smalltalk: Who Knows Where the Time Goes?", in *Smalltalk-80: Bits of History, Words of Advice*, G. Krasner (editor), September, 1983.
[Ungar84a]	David Ungar, Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm, *ACM Software Eng. Notes/SIGPLAN Notices Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, PA, April, 1984.
[Ungar84b]	David Ungar, Ricki Blau, Peter Foley, A. Dain Samples, and David Patterson, "Architecture of SOAR: Smalltalk on a RISC", 11th Annual International Symposium on Computer Architecture, Ann Arbor, Michigan, June 4-7, 1984.

[Ungar86] David Ungar, "The Design and Evaluation of A High Performance Smalltalk System", PhD thesis, UCB, 1986; issued as tech. rpt. UCB/CSD 86/287.

[Wallace83] Dave Wallace, "Making Smalltalk less Becoming: Removing Primitive Becomes from Smalltalk-80", in *Smalltalk on a RISC, Architectural Investigations,* Proceedings of CS292R, April, 1983.

[Waring] *Your Waring Cookbook: The Pleasure of Blending (For the 14-speed blender),* Waring Food Corporation, [undated], p 1.